GOOGLE'S MOTION TO STRIKE PORTIONS OF THE MITCHELL PATENT REPORT

CIV. NO. CV 10-03561-WHA

Exhibit C

The '104 Reissue Patent	Infringed By
	• Bytecode verification is necessary, but slow, so we want to verify as much as possible outside app execution. • Bytecode optimization (quickened instructions, method pruning) is important for speed and battery life. • For security reasons, processes may not edit shared code. The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off disk in many small pieces). On the other hand, having the bytecode on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations. The goals led us to make some fundamental decisions: • Multiple classes are aggregated into a single "DEX" file. • DEX files are mapped read-only and shared between processes. • Byte ordering and word alignment are adjusted to suit the local system. • Bytecode verification is mandatory for all classes, but we want to "pre-verify" whatever we can. • Optimizations that require rewriting bytecode must be done ahead of time. • The consequences of these decisions are explained in the following sections. ….
[11-b] and a processor configured to execute said instructions containing one or more symbolic references by determining a numerical reference corresponding to said symbolic reference, storing said	Any device running Android has a processor configured to execute said instructions containing one or more symbolic references by determining a numerical reference corresponding to said symbolic reference, storing said numerical references, and obtaining data in accordance to said numerical references. *See, e.g.*, \dalvik\vm\oo\Resolve.h: /* * Resolve "constant pool" references into pointers to VM structs. */

The '104 Reissue Patent	Infringed By
numerical references, and obtaining data in accordance to said numerical references.	#ifndef _DALVIK_OO_RESOLVE #define _DALVIK_OO_RESOLVE /* * "Direct" and "virtual" methods are stored independently. The type of call * used to invoke the method determines which list we search, and whether * we travel up into superclasses. * * (<clinit>, <init>, and methods declared "private" or "static" are stored * in the "direct" list. All others are stored in the "virtual" list.) */ typedef enum MethodType { METHOD_UNKNOWN = 0, METHOD_DIRECT, // <init>, private METHOD_STATIC, // static METHOD_VIRTUAL, // virtual, super METHOD_INTERFACE // interface } MethodType; /* * Resolve a class, given the referring class and a constant pool index * for the DexTypeId. * * Does not initialize the class. * * Throws an exception and returns NULL on failure. */ ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool fromUnverifiedConstant); /* * Resolve a direct, static, or virtual method.

The '104 Reissue Patent	Infringed By
	* * Can cause the method's class to be initialized if methodType is * METHOD_STATIC. * * Throws an exception and returns NULL on failure. */ Method* dvmResolveMethod(const ClassObject* referrer, u4 methodIdx, MethodType methodType); /* * Resolve an interface method. * * Throws an exception and returns NULL on failure. */ Method* dvmResolveInterfaceMethod(const ClassObject* referrer, u4 methodIdx); /* * Resolve an instance field. * * Throws an exception and returns NULL on failure. */ InstField* dvmResolveInstField(const ClassObject* referrer, u4 ifieldIdx); /* * Resolve a static field. * * Causes the field's class to be initialized. * * Throws an exception and returns NULL on failure. */ StaticField* dvmResolveStaticField(const ClassObject* referrer, u4 sfieldIdx);

The '104 Reissue Patent	Infringed By
	/* * Resolve a "const-string" reference. * * Throws an exception and returns NULL on failure. */ StringObject* dvmResolveString(const ClassObject* referrer, u4 stringIdx); /* * Return debug string constant for enum. */ const char* dvmMethodTypeStr(MethodType methodType); #endif /*_DALVIK_OO_RESOLVE*/ \dalvik\vm\oo\Resolve.c: /* * Resolve classes, methods, fields, and strings. * * According to the VM spec (v2 5.5), classes may be initialized by use * of the "new", "getstatic", "putstatic", or "invokestatic" instructions. * If we are resolving a static method or static field, we make the * initialization check here. * * (NOTE: the verifier has its own resolve functions, which can be invoked * if a class isn't pre-verified. Those functions must not update the * "resolved stuff" tables for static fields and methods, because they do * not perform initialization.) */ #include "Dalvik.h" #include <stdlib.h>

The '104 Reissue Patent	Infringed By
	/* * Find the class corresponding to "classIdx", which maps to a class name * string. It might be in the same DEX file as "referrer", in a different * DEX file, generated by a class loader, or generated by the VM (e.g. * array classes). * * Because the DexTypeId is associated with the referring class' DEX file, * we may have to resolve the same class more than once if it's referred * to from classes in multiple DEX files. This is a necessary property for * DEX files associated with different class loaders. * * We cache a copy of the lookup in the DexFile's "resolved class" table, * so future references to "classIdx" are faster. * * Note that "referrer" may be in the process of being linked. * * Traditional VMs might do access checks here, but in Dalvik the class * "constant pool" is shared between all classes in the DEX file. We rely * on the verifier to do the checks for us. * * Does not initialize the class. * * "fromUnverifiedConstant" should only be set if this call is the direct * result of executing a "const-class" or "instance-of" instruction, which * use class constants not resolved by the bytecode verifier. * * Returns NULL with an exception raised on failure. */ ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool fromUnverifiedConstant)

The '104 Reissue Patent	Infringed By

```
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const char* className;

    /*
     * Check the table first -- this gets called from the other "resolve"
     * methods.
     */
    resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
    if (resClass != NULL)
        return resClass;

    LOGVV("--- resolving class %u (referrer=%s cl=%p)\n",
        classIdx, referrer->descriptor, referrer->classLoader);

    /*
     * Class hasn't been loaded yet, or is in the process of being loaded
     * and initialized now.  Try to get a copy.  If we find one, put the
     * pointer in the DexTypeId.  There isn't a race condition here --
     * 32-bit writes are guaranteed atomic on all target platforms.  Worst
     * case we have two threads storing the same value.
     *
     * If this is an array class, we'll generate it here.
     */
    className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
    if (className[0] != '\0' && className[1] == '\0') {
        /* primitive type */
        resClass = dvmFindPrimitiveClass(className[0]);
    } else {
        resClass = dvmFindClassNoInit(className, referrer->classLoader);
    }
```

The '104 Reissue Patent	Infringed By

```
if (resClass != NULL) {
    /*
     * If the referrer was pre-verified, the resolved class must come
     * from the same DEX or from a bootstrap class.  The pre-verifier
     * makes assumptions that could be invalidated by a wacky class
     * loader.  (See the notes at the top of oo/Class.c.)
     *
     * The verifier does *not* fail a class for using a const-class
     * or instance-of instruction referring to an unresolveable class,
     * because the result of the instruction is simply a Class object
     * or boolean -- there's no need to resolve the class object during
     * verification.  Instance field and virtual method accesses can
     * break dangerously if we get the wrong class, but const-class and
     * instance-of are only interesting at execution time.  So, if we
     * we got here as part of executing one of the "unverified class"
     * instructions, we skip the additional check.
     *
     * Ditto for class references from annotations and exception
     * handler lists.
     */
    if (!fromUnverifiedConstant &&
        IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
    {
        ClassObject* resClassCheck = resClass;
        if (dvmIsArrayClass(resClassCheck))
            resClassCheck = resClassCheck->elementClass;

        if (referrer->pDvmDex != resClassCheck->pDvmDex &&
            resClassCheck->classLoader != NULL)
        {
            LOGW("Class resolved by unexpected DEX:"
```

The '104 Reissue Patent	Infringed By

```
                    " %s(%p):%p ref [%s] %s(%p):%p\n",
                    referrer->descriptor, referrer->classLoader,
                    referrer->pDvmDex,
                    resClass->descriptor, resClassCheck->descriptor,
                    resClassCheck->classLoader, resClassCheck->pDvmDex);
                LOGW("(%s had used a different %s during pre-verification)\n",
                    referrer->descriptor, resClass->descriptor);
                dvmThrowException("Ljava/lang/IllegalAccessError;",
                    "Class ref in pre-verified class resolved to unexpected "
                    "implementation");
                return NULL;
            }
        }

        LOGVV("##### +ResolveClass(%s): referrer=%s dex=%p ldr=%p ref=%d\n",
            resClass->descriptor, referrer->descriptor, referrer->pDvmDex,
            referrer->classLoader, classIdx);

        /*
         * Add what we found to the list so we can skip the class search
         * next time through.
         *
         * TODO: should we be doing this when fromUnverifiedConstant==true?
         * (see comments at top of oo/Class.c)
         */
        dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
    } else {
        /* not found, exception should be raised */
        LOGVV("Class not found: %s\n",
            dexStringByTypeIdx(pDvmDex->pDexFile, classIdx));
        assert(dvmCheckException(dvmThreadSelf()));
    }
```

The '104 Reissue Patent	Infringed By

```
    return resClass;
}


/*
 * Find the method corresponding to "methodRef".
 *
 * We use "referrer" to find the DexFile with the constant pool that
 * "methodRef" is an index into.  We also use its class loader.  The method
 * being resolved may very well be in a different DEX file.
 *
 * If this is a static method, we ensure that the method's class is
 * initialized.
 */
Method* dvmResolveMethod(const ClassObject* referrer, u4 methodIdx,
    MethodType methodType)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const DexMethodId* pMethodId;
    Method* resMethod;

    assert(methodType != METHOD_INTERFACE);

    LOGVV("--- resolving method %u (referrer=%s)\n", methodIdx,
        referrer->descriptor);
    pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

    resClass = dvmResolveClass(referrer, pMethodId->classIdx, false);
    if (resClass == NULL) {
        /* can't find the class that the method is a part of */
```

The '104 Reissue Patent	Infringed By

```
    assert(dvmCheckException(dvmThreadSelf()));
    return NULL;
}
if (dvmIsInterfaceClass(resClass)) {
    /* method is part of an interface */
    dvmThrowExceptionWithClassMessage(
        "Ljava/lang/IncompatibleClassChangeError;",
        resClass->descriptor);
    return NULL;
}

const char* name = dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx);
DexProto proto;
dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);

/*
 * We need to chase up the class hierarchy to find methods defined
 * in super-classes.  (We only want to check the current class
 * if we're looking for a constructor; since DIRECT calls are only
 * for constructors and private methods, we don't want to walk up.)
 */
if (methodType == METHOD_DIRECT) {
    resMethod = dvmFindDirectMethod(resClass, name, &proto);
} else if (methodType == METHOD_STATIC) {
    resMethod = dvmFindDirectMethodHier(resClass, name, &proto);
} else {
    resMethod = dvmFindVirtualMethodHier(resClass, name, &proto);
}

if (resMethod == NULL) {
    dvmThrowException("Ljava/lang/NoSuchMethodError;", name);
    return NULL;
```

The '104 Reissue Patent	Infringed By

```
}

LOGVV("--- found method %d (%s.%s)\n",
    methodIdx, resClass->descriptor, resMethod->name);

/* see if this is a pure-abstract method */
if (dvmIsAbstractMethod(resMethod) && !dvmIsAbstractClass(resClass)) {
    dvmThrowException("Ljava/lang/AbstractMethodError;", name);
    return NULL;
}

/*
 * If we're the first to resolve this class, we need to initialize
 * it now.  Only necessary for METHOD_STATIC.
 */
if (methodType == METHOD_STATIC) {
    if (!dvmIsClassInitialized(resMethod->clazz) &&
        !dvmInitClass(resMethod->clazz))
    {
        assert(dvmCheckException(dvmThreadSelf()));
        return NULL;
    } else {
        assert(!dvmCheckException(dvmThreadSelf()));
    }
} else {
    /*
     * Edge case: if the <clinit> for a class creates an instance
     * of itself, we will call <init> on a class that is still being
     * initialized by us.
     */
    assert(dvmIsClassInitialized(resMethod->clazz) ||
           dvmIsClassInitializing(resMethod->clazz));
```

The '104 Reissue Patent	Infringed By

```
    }

    /*
     * The class is initialized, the method has been found.  Add a pointer
     * to our data structure so we don't have to jump through the hoops again.
     */
    dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);

    return resMethod;
}

/*
 * Resolve an interface method reference.
 *
 * Returns NULL with an exception raised on failure.
 */
Method* dvmResolveInterfaceMethod(const ClassObject* referrer, u4 methodIdx)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const DexMethodId* pMethodId;
    Method* resMethod;
    int i;

    LOGVV("--- resolving interface method %d (referrer=%s)\n",
        methodIdx, referrer->descriptor);
    pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

    resClass = dvmResolveClass(referrer, pMethodId->classIdx, false);
    if (resClass == NULL) {
        /* can't find the class that the method is a part of */
        assert(dvmCheckException(dvmThreadSelf()));
```

The '104 Reissue Patent	Infringed By

```
    return NULL;
}
if (!dvmIsInterfaceClass(resClass)) {
    /* whoops */
    dvmThrowExceptionWithClassMessage(
        "Ljava/lang/IncompatibleClassChangeError;",
        resClass->descriptor);
    return NULL;
}

/*
 * This is the first time the method has been resolved.  Set it in our
 * resolved-method structure.  It always resolves to the same thing,
 * so looking it up and storing it doesn't create a race condition.
 *
 * If we scan into the interface's superclass -- which is always
 * java/lang/Object -- we will catch things like:
 *   interface I ...
 *   I myobj = (something that implements I)
 *   myobj.hashCode()
 * However, the Method->methodIndex will be an offset into clazz->vtable,
 * rather than an offset into clazz->iftable.  The invoke-interface
 * code can test to see if the method returned is abstract or concrete,
 * and use methodIndex accordingly.  I'm not doing this yet because
 * (a) we waste time in an unusual case, and (b) we're probably going
 * to fix it in the DEX optimizer.
 *
 * We do need to scan the superinterfaces, in case we're invoking a
 * superinterface method on an interface reference.  The class in the
 * DexTypeId is for the static type of the object, not the class in
 * which the method is first defined.  We have the full, flattened
 * list in "iftable".
```

The '104 Reissue Patent	Infringed By

```
 */
const char* methodName =
    dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx);

DexProto proto;
dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);

LOGVV("+++ looking for '%s' '%s' in resClass='%s'\n",
    methodName, methodSig, resClass->descriptor);
resMethod = dvmFindVirtualMethod(resClass, methodName, &proto);
if (resMethod == NULL) {
    LOGVV("+++ did not resolve immediately\n");
    for (i = 0; i < resClass->iftableCount; i++) {
        resMethod = dvmFindVirtualMethod(resClass->iftable[i].clazz,
                methodName, &proto);
        if (resMethod != NULL)
            break;
    }

    if (resMethod == NULL) {
        dvmThrowException("Ljava/lang/NoSuchMethodError;", methodName);
        return NULL;
    }
} else {
    LOGVV("+++ resolved immediately: %s (%s %d)\n", resMethod->name,
        resMethod->clazz->descriptor, (u4) resMethod->methodIndex);
}

LOGVV("--- found interface method %d (%s.%s)\n",
    methodIdx, resClass->descriptor, resMethod->name);

/* we're expecting this to be abstract */
```

The '104 Reissue Patent	Infringed By

```
    assert(dvmIsAbstractMethod(resMethod));

    /* interface methods are always public; no need to check access */

    /*
     * The interface class *may* be initialized.  According to VM spec
     * v2 2.17.4, the interfaces a class refers to "need not" be initialized
     * when the class is initialized.
     *
     * It isn't necessary for an interface class to be initialized before
     * we resolve methods on that interface.
     *
     * We choose not to do the initialization now.
     */
    //assert(dvmIsClassInitialized(resMethod->clazz));

    /*
     * The class is initialized, the method has been found.  Add a pointer
     * to our data structure so we don't have to jump through the hoops again.
     */
    dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);

    return resMethod;
}

/*
 * Resolve an instance field reference.
 *
 * Returns NULL and throws an exception on error (no such field, illegal
 * access).
 */
InstField* dvmResolveInstField(const ClassObject* referrer, u4 ifieldIdx)
```

The '104 Reissue Patent	Infringed By

```
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const DexFieldId* pFieldId;
    InstField* resField;

    LOGVV("--- resolving field %u (referrer=%s cl=%p)\n",
        ifieldIdx, referrer->descriptor, referrer->classLoader);

    pFieldId = dexGetFieldId(pDvmDex->pDexFile, ifieldIdx);

    /*
     * Find the field's class.
     */
    resClass = dvmResolveClass(referrer, pFieldId->classIdx, false);
    if (resClass == NULL) {
        assert(dvmCheckException(dvmThreadSelf()));
        return NULL;
    }

    resField = dvmFindInstanceFieldHier(resClass,
        dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
        dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
    if (resField == NULL) {
        dvmThrowException("Ljava/lang/NoSuchFieldError;",
            dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
        return NULL;
    }

    /*
     * Class must be initialized by now (unless verifier is buggy).  We
     * could still be in the process of initializing it if the field
```

The '104 Reissue Patent	Infringed By

```
     * access is from a static initializer.
     */
    assert(dvmIsClassInitialized(resField->field.clazz) ||
           dvmIsClassInitializing(resField->field.clazz));

    /*
     * The class is initialized, the method has been found.  Add a pointer
     * to our data structure so we don't have to jump through the hoops again.
     */
    dvmDexSetResolvedField(pDvmDex, ifieldIdx, (Field*)resField);
    LOGVV("    field %u is %s.%s\n",
        ifieldIdx, resField->field.clazz->descriptor, resField->field.name);

    return resField;
}

/*
 * Resolve a static field reference.  The DexFile format doesn't distinguish
 * between static and instance field references, so the "resolved" pointer
 * in the Dex struct will have the wrong type.  We trivially cast it here.
 *
 * Causes the field's class to be initialized.
 */
StaticField* dvmResolveStaticField(const ClassObject* referrer, u4 sfieldIdx)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const DexFieldId* pFieldId;
    StaticField* resField;

    pFieldId = dexGetFieldId(pDvmDex->pDexFile, sfieldIdx);
```

The '104 Reissue Patent	Infringed By

```
/*
 * Find the field's class.
 */
resClass = dvmResolveClass(referrer, pFieldId->classIdx, false);
if (resClass == NULL) {
    assert(dvmCheckException(dvmThreadSelf()));
    return NULL;
}

resField = dvmFindStaticFieldHier(resClass,
            dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
            dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
if (resField == NULL) {
    dvmThrowException("Ljava/lang/NoSuchFieldError;",
        dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
    return NULL;
}

/*
 * If we're the first to resolve the field in which this class resides,
 * we need to do it now.  Note that, if the field was inherited from
 * a superclass, it is not necessarily the same as "resClass".
 */
if (!dvmIsClassInitialized(resField->field.clazz) &&
    !dvmInitClass(resField->field.clazz))
{
    assert(dvmCheckException(dvmThreadSelf()));
    return NULL;
}

/*
 * The class is initialized, the method has been found.  Add a pointer
```

The '104 Reissue Patent	Infringed By

```
     * to our data structure so we don't have to jump through the hoops again.
     */
    dvmDexSetResolvedField(pDvmDex, sfieldIdx, (Field*) resField);

    return resField;
}


/*
 * Resolve a string reference.
 *
 * Finding the string is easy.  We need to return a reference to a
 * java/lang/String object, not a bunch of characters, which means the
 * first time we get here we need to create an interned string.
 */
StringObject* dvmResolveString(const ClassObject* referrer, u4 stringIdx)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    StringObject* strObj;
    StringObject* internStrObj;
    const char* utf8;
    u4 utf16Size;

    LOGVV("+++ resolving string, referrer is %s\n", referrer->descriptor);

    /*
     * Create a UTF-16 version so we can trivially compare it to what's
     * already interned.
     */
    utf8 = dexStringAndSizeById(pDvmDex->pDexFile, stringIdx, &utf16Size);
    strObj = dvmCreateStringFromCstrAndLength(utf8, utf16Size,
            ALLOC_DEFAULT);
```

The '104 Reissue Patent	Infringed By

```
    if (strObj == NULL) {
        /* ran out of space in GC heap? */
        assert(dvmCheckException(dvmThreadSelf()));
        goto bail;
    }

    /*
     * Add it to the intern list.  The return value is the one in the
     * intern list, which (due to race conditions) may or may not be
     * the one we just created.  The intern list is synchronized, so
     * there will be only one "live" version.
     *
     * By requesting an immortal interned string, we guarantee that
     * the returned object will never be collected by the GC.
     *
     * A NULL return here indicates some sort of hashing failure.
     */
    internStrObj = dvmLookupImmortalInternedString(strObj);
    dvmReleaseTrackedAlloc((Object*) strObj, NULL);
    strObj = internStrObj;
    if (strObj == NULL) {
        assert(dvmCheckException(dvmThreadSelf()));
        goto bail;
    }

    /* save a reference so we can go straight to the object next time */
    dvmDexSetResolvedString(pDvmDex, stringIdx, strObj);

bail:
    return strObj;
}
```

The '104 Reissue Patent	Infringed By

```
/*
 * For debugging: return a string representing the methodType.
 */
const char* dvmMethodTypeStr(MethodType methodType)
{
    switch (methodType) {
    case METHOD_DIRECT:         return "direct";
    case METHOD_STATIC:         return "static";
    case METHOD_VIRTUAL:        return "virtual";
    case METHOD_INTERFACE:      return "interface";
    case METHOD_UNKNOWN:        return "UNKNOWN";
    }
    assert(false);
    return "BOGUS";
}
```

Another way that Android and Android-based devices meet the claim limitation is through the dexopt tool.

See, e.g., dalvik\docs\dexopt.html; *see also*, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html:

dexopt

We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in.

The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On

The '104 Reissue Patent	Infringed By
	completion, the process exits, freeing all resources. It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once. …. *See also, e.g.*, dalvik\docs\ embedded-vm-control.html#verifier ("The system tries to pre-verify all classes in a DEX file to reduce class load overhead, and performs a series of optimizations to improve runtime performance. Both of these are done by the dexopt command, either in the build system or by the installer. On a development device, dexopt may be run the first time a DEX file is used and whenever it or one of its dependencies is updated ("just-in-time" optimization and verification)."). Dexopt loads the intermediate code class files, and when it encounters a symbolic reference (e.g., virtual method calls, field gets/puts), it determines the numerical reference corresponding to the symbolic reference and stores the numerical reference so that the processor can obtain data in accordance to the numerical references. *See, e.g.*, dalvik\docs\dexopt.html; *see also*, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html: **Dalvik Optimization and Verification With *dexopt*** The Dalvik virtual machine was designed specifically for the Android mobile platform. The target systems have little RAM, store data on slow internal flash memory, and generally have the performance characteristics of decade-old desktop systems. They also run Linux, which provides virtual memory, processes and threads, and UID-based security mechanisms. The features and limitations caused us to focus on certain goals: • Class data, notably bytecode, must be shared between multiple processes to minimize total system memory usage. • The overhead in launching a new app must be minimized to keep the device responsive. • Storing class data in individual files results in a lot of redundancy, especially with respect

The '104 Reissue Patent	Infringed By
	to strings. To conserve disk space we need to factor this out. • Parsing class data fields adds unnecessary overhead during class loading. Accessing data values (e.g. integers and strings) directly as C types is better. • Bytecode verification is necessary, but slow, so we want to verify as much as possible outside app execution. • Bytecode optimization (quickened instructions, method pruning) is important for speed and battery life. • For security reasons, processes may not edit shared code. The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off disk in many small pieces). On the other hand, having the bytecode on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations. The goals led us to make some fundamental decisions: • Multiple classes are aggregated into a single "DEX" file. • DEX files are mapped read-only and shared between processes. • Byte ordering and word alignment are adjusted to suit the local system. • Bytecode verification is mandatory for all classes, but we want to "pre-verify" whatever we can. • Optimizations that require rewriting bytecode must be done ahead of time. • The consequences of these decisions are explained in the following sections. …. **dexopt** We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in.

The '104 Reissue Patent	Infringed By
	The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources. It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once. …. **Optimization** Virtual machine interpreters typically perform certain optimizations the first time a piece of code is used. Constant pool references are replaced with pointers to internal data structures, operations that always succeed or always work a certain way are replaced with simpler forms. Some of these require information only available at runtime, others can be inferred statically when certain assumptions are made. The Dalvik optimizer does the following: • For virtual method calls, replace the method index with a vtable index. • For instance field get/put, replace the field index with a byte offset. Also, merge the boolean / byte / char / short variants into a single 32-bit form (less code in the interpreter means more room in the CPU I-cache). • Replace a handful of high-volume calls, like String.length(), with "inline" replacements. This skips the usual method call overhead, directly switching from the interpreter to a native implementation. • Prune empty methods. The simplest example is Object.<init>, which does nothing, but must be called whenever any object is allocated. The instruction is replaced with a new version that acts as a no-op unless a debugger is attached. • Append pre-computed data. For example, the VM wants to have a hash table for lookups on class name. Instead of computing this when the DEX file is loaded, we can compute it now, saving heap space and computation time in every VM where the DEX is loaded.

The '104 Reissue Patent	Infringed By
	All of the instruction modifications involve replacing the opcode with one not defined by the Dalvik specification. This allows us to freely mix optimized and unoptimized instructions. The set of optimized instructions, and their exact representation, is tied closely to the VM version. Most of the optimizations are obvious "wins". The use of raw indices and offsets not only allows us to execute more quickly, we can also skip the initial symbolic resolution. Pre-computation eats up disk space, and so must be done in moderation. There are a couple of potential sources of trouble with these optimizations. First, vtable indices and byte offsets are subject to change if the VM is updated. Second, if a superclass is in a different DEX, and that other DEX is updated, we need to ensure that our optimized indices and offsets are updated as well. A similar but more subtle problem emerges when user-defined class loaders are employed: the class we actually call may not be the one we expected to call. These problems are addressed with dependency lists and some limitations on what can be optimized. *See, e.g.*, dalvik\vm\analysis\ReduceConstants.c: /* Overview When a class, method, field, or string constant is referred to from Dalvik bytecode, the reference takes the form of an integer index value. This value indexes into an array of type_id_item, method_id_item, field_id_item, or string_id_item in the DEX file. The first three themselves contain (directly or indirectly) indexes to strings that the resolver uses to convert the instruction stream index into a pointer to the appropriate object or struct. For example, an invoke-virtual instruction needs to specify which method is to be invoked. The method constant indexes into the method_id_item

The '104 Reissue Patent	Infringed By
	array, each entry of which has indexes that specify the defining class (type_id_item), method name (string_id_item), and method prototype (proto_id_item). The type_id_item just holds an index to a string_id_item, which holds the file offset to the string with the class name. The VM finds the class by name, then searches through the class' table of virtual methods to find one with a matching name and prototype. This process is fairly expensive, so after the first time it completes successfully, the VM records that the method index resolved to a specific Method struct. On subsequent execution, the VM just pulls the Method ptr out of the resolved-methods array. A similar approach is used with the indexes for classes, fields, and string constants. The problem with this approach is that we need to have a "resolved" entry for every possible class, method, field, and string constant in every DEX file, even if some of those aren't used from code. The DEX string constant table has entries for method prototypes and class names that are never used by the code, and "public static final" fields often turn into immediate constants. The resolution table entries are only 4 bytes each, but there are roughly 200,000 of them in the bootstrap classes alone. DEX optimization removes many index references by replacing virtual method indexes with vtable offsets and instance field indexes with byte offsets. In the earlier example, the method would be resolved at "dexopt" time, and the instruction rewritten as invoke-virtual-quick with the vtable offset. (There are comparatively few classes compared to other constant pool entries, and a much higher percentage (typically 60-70%) are used. The biggest gains come from the string pool.) Using the resolved-entity tables provides a substantial performance improvement, but results in applications allocating 1MB+ of tables that

The '104 Reissue Patent	Infringed By
	are 70% unused. The used and unused entries are freely intermixed, preventing effective sharing with the zygote process, and resulting in large numbers of private/dirty pages on the native heap as the tables populate on first use. The trick is to reduce the memory usage without decreasing performance. Using smaller resolved-entity tables can actually give us a speed boost, because we'll have a smaller "live" set of pages and make more effective use of the data cache. The approach we're going to use is to determine the set of indexes that could potentially be resolved, generate a mapping from the minimal set to the full set, and append the mapping to the DEX file. This is done at "dexopt" time, because we need to keep the changes in shared/read-only pages or we'll lose the benefits of doing the work. There are two ways to create and use the new mapping: (1) Write the entire full->minimal mapping to the ".odex" file. On every instruction that uses an index, use the mapping to determine the "compressed" constant value, and then use that to index into the resolved-entity tables on the heap. The instruction stream is unchanged, and the resolver can easily tell if a given index is cacheable. (2) Write the inverse miminal->full mapping to the ".odex" file, and rewrite the constants in the instruction stream. The interpreter is unchanged, and the resolver code uses the mapping to find the original data in the DEX. Approach #1 is easier and safer to implement, but it requires a table lookup every time we execute an instruction that includes a constant

The '104 Reissue Patent	Infringed By
	pool reference. This causes an unacceptable performance hit, chiefly because we're hitting semi-random memory pages and hosing the data cache. This is mitigated somewhat by DEX optimizations that replace the constant with a vtable index or field byte offset. Approach #1 also requires a larger map table, increasing the size of the DEX on disk. One nice property of approach #1 is that most of the DEX file is unmodified, so use of the mapping is a runtime decision. Approach #2 is preferred for performance reasons. The class/method/field/string resolver code has to handle indices from three sources: interpreted instructions, annotations, and exception "catch" lists. Sometimes these occur indirectly, e.g. we need to resolve the declaring class associated with fields and methods when the latter two are themselves resolved. Parsing and rewriting instructions is fairly straightforward, but annotations use a complex format with variable-width index values. We can safely rewrite index values in annotations if we guarantee that the new value is smaller than the original. This implies a two-pass approach: the first determines the set of indexes actually used, the second does the rewrite. Doing the rewrite in a single pass would be much harder. Instances of the "original" indices will still be found in the file; if we try to be all-inclusive we will include some stuff that doesn't need to be there (e.g. we don't generally need to cache the class name string index result, since once we have the class resolved we don't need to look it up by name through the resolver again). There is some potential for performance improvement by caching more than we strictly need, but we can afford to give up a little performance during class loading if it allows us to regain some memory.

The '104 Reissue Patent	Infringed By
	For safety and debugging, it's useful to distinguish the "compressed" constants in some way, e.g. setting the high bit when we rewrite them. In practice we don't have any free bits: indexes are usually 16-bit values, and we have more than 32,767 string constants in at least one of our core DEX files. Also, this does not work with constants embedded in annotations, because of the variable-width encoding. We should be safe if we can establish a clear distinction between sources of "original" and "compressed" indices. If the values get crossed up we can end up with elusive bugs. The easiest approach is to declare that only indices pulled from certain locations (the instruction stream and/or annotations) are compressed. This prevents us from adding indices in arbitrary locations to the compressed set, but should allow a reasonably robust implementation. … */ dalvik\vm\analysis\DexOptimize.h (similarly in dalvik\vm\analysis\Optimize.h): /* * Abbreviated resolution functions, for use by optimization and verification * code. */ ClassObject* dvmOptResolveClass(ClassObject* referrer, u4 classIdx, VerifyError* pFailure); Method* dvmOptResolveMethod(ClassObject* referrer, u4 methodIdx, MethodType methodType, VerifyError* pFailure); Method* dvmOptResolveInterfaceMethod(ClassObject* referrer, u4 methodIdx); InstField* dvmOptResolveInstField(ClassObject* referrer, u4 ifieldIdx, VerifyError* pFailure); StaticField* dvmOptResolveStaticField(ClassObject* referrer, u4 sfieldIdx,

The '104 Reissue Patent	Infringed By

	VerifyError* pFailure);
	
	dalvik\vm\analysis\DexOptimize.c (similarly in dalvik\vm\analysis\Optimize.c):
	/*
	*
	==========
	======
	* Optimizations
	*
	==========
	======
	*/
	
	/*
	* Perform in-place rewrites on a memory-mapped DEX file.
	*
	* This happens in a short-lived child process, so we can go nutty with
	* loading classes and allocating memory.
	*/
	static bool rewriteDex(u1* addr, int len, bool doVerify, bool doOpt,
	u4* pHeaderFlags, DexClassLookup** ppClassLookup)
	{
	u8 prepWhen, loadWhen, verifyWhen, optWhen;
	DvmDex* pDvmDex = NULL;
	bool result = false;
	
	*pHeaderFlags = 0;
	
	LOGV("+++ swapping bytes\n");
	if (dexFixByteOrdering(addr, len) != 0)
	goto bail;
	#if __BYTE_ORDER != __LITTLE_ENDIAN

The '104 Reissue Patent	Infringed By

```
    *pHeaderFlags |= DEX_OPT_FLAG_BIG;
#endif

    /*
     * Now that the DEX file can be read directly, create a DexFile for it.
     */
    if (dvmDexFileOpenPartial(addr, len, &pDvmDex) != 0) {
        LOGE("Unable to create DexFile\n");
        goto bail;
    }

    /*
     * Create the class lookup table.
     */
    //startWhen = dvmGetRelativeTimeUsec();
    *ppClassLookup = dexCreateClassLookup(pDvmDex->pDexFile);
    if (*ppClassLookup == NULL)
        goto bail;

    /*
     * Bail out early if they don't want The Works.  The current implementation
     * doesn't fork a new process if this flag isn't set, so we really don't
     * want to continue on with the crazy class loading.
     */
    if (!doVerify && !doOpt) {
        result = true;
        goto bail;
    }

    /* this is needed for the next part */
    pDvmDex->pDexFile->pClassLookup = *ppClassLookup;
```

The '104 Reissue Patent	Infringed By

	prepWhen = dvmGetRelativeTimeUsec();
	
	/*
	* Load all classes found in this DEX file. If they fail to load for
	* some reason, they won't get verified (which is as it should be).
	*/
	if (!loadAllClasses(pDvmDex))
	goto bail;
	loadWhen = dvmGetRelativeTimeUsec();
	
	/*
	* Verify all classes in the DEX file. Export the "is verified" flag
	* to the DEX file we're creating.
	*/
	if (doVerify) {
	dvmVerifyAllClasses(pDvmDex->pDexFile);
	*pHeaderFlags |= DEX_FLAG_VERIFIED;
	}
	verifyWhen = dvmGetRelativeTimeUsec();
	
	/*
	* Optimize the classes we successfully loaded. If the opt mode is
	* OPTIMIZE_MODE_VERIFIED, each class must have been successfully
	* verified or we'll skip it.
	*/
	#ifndef PROFILE_FIELD_ACCESS
	if (doOpt) {
	optimizeLoadedClasses(pDvmDex->pDexFile);
	*pHeaderFlags |= DEX_OPT_FLAG_FIELDS | DEX_OPT_FLAG_INVOCATIONS;
	}
	#endif
	optWhen = dvmGetRelativeTimeUsec();

The '104 Reissue Patent	Infringed By

	LOGD("DexOpt: load %dms, verify %dms, opt %dms\n",
	(int) (loadWhen - prepWhen) / 1000,
	(int) (verifyWhen - loadWhen) / 1000,
	(int) (optWhen - verifyWhen) / 1000);
	
	result = true;
	
	bail:
	/* free up storage */
	dvmDexFileFree(pDvmDex);
	
	return result;
	}
	
	…
	
	/*
	* Alternate version of dvmResolveClass for use with verification and
	* optimization. Performs access checks on every resolve, and refuses
	* to acknowledge the existence of classes defined in more than one DEX
	* file.
	*
	* Exceptions caused by failures are cleared before returning.
	*
	* On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
	*/
	ClassObject* dvmOptResolveClass(ClassObject* referrer, u4 classIdx,
	VerifyError* pFailure)
	{
	DvmDex* pDvmDex = referrer->pDvmDex;

The '104 Reissue Patent	Infringed By

```
ClassObject* resClass;

/*
 * Check the table first.  If not there, do the lookup by name.
 */
resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
if (resClass == NULL) {
    const char* className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
    if (className[0] != '\0' && className[1] == '\0') {
        /* primitive type */
        resClass = dvmFindPrimitiveClass(className[0]);
    } else {
        resClass = dvmFindClassNoInit(className, referrer->classLoader);
    }
    if (resClass == NULL) {
        /* not found, exception should be raised */
        LOGV("DexOpt: class %d (%s) not found\n",
            classIdx,
            dexStringByTypeIdx(pDvmDex->pDexFile, classIdx));
        if (pFailure != NULL) {
            /* dig through the wrappers to find the original failure */
            Object* excep = dvmGetException(dvmThreadSelf());
            while (true) {
                Object* cause = dvmGetExceptionCause(excep);
                if (cause == NULL)
                    break;
                excep = cause;
            }
            if (strcmp(excep->clazz->descriptor,
                "Ljava/lang/IncompatibleClassChangeError;") == 0)
            {
                *pFailure = VERIFY_ERROR_CLASS_CHANGE;
```

The '104 Reissue Patent	Infringed By

```
            } else {
                *pFailure = VERIFY_ERROR_NO_CLASS;
            }
        }
        dvmClearOptException(dvmThreadSelf());
        return NULL;
    }

    /*
     * Add it to the resolved table so we're faster on the next lookup.
     */
    dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
}

/* multiple definitions? */
if (IS_CLASS_FLAG_SET(resClass, CLASS_MULTIPLE_DEFS)) {
    LOGI("DexOpt: not resolving ambiguous class '%s'\n",
        resClass->descriptor);
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_NO_CLASS;
    return NULL;
}

/* access allowed? */
tweakLoader(referrer, resClass);
bool allowed = dvmCheckClassAccess(referrer, resClass);
untweakLoader(referrer, resClass);
if (!allowed) {
    LOGW("DexOpt: resolve class illegal access: %s -> %s\n",
        referrer->descriptor, resClass->descriptor);
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_ACCESS_CLASS;
```

The '104 Reissue Patent	Infringed By

```
        return NULL;
    }

    return resClass;
}

/*
 * Alternate version of dvmResolveInstField().
 *
 * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
 */
InstField* dvmOptResolveInstField(ClassObject* referrer, u4 ifieldIdx,
    VerifyError* pFailure)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    InstField* resField;

    resField = (InstField*) dvmDexGetResolvedField(pDvmDex, ifieldIdx);
    if (resField == NULL) {
        const DexFieldId* pFieldId;
        ClassObject* resClass;

        pFieldId = dexGetFieldId(pDvmDex->pDexFile, ifieldIdx);

        /*
         * Find the field's class.
         */
        resClass = dvmOptResolveClass(referrer, pFieldId->classIdx, pFailure);
        if (resClass == NULL) {
            //dvmClearOptException(dvmThreadSelf());
            assert(!dvmCheckException(dvmThreadSelf()));
            if (pFailure != NULL) { assert(!VERIFY_OK(*pFailure)); }
```

The '104 Reissue Patent	Infringed By

```
        return NULL;
    }

    resField = (InstField*)dvmFindFieldHier(resClass,
        dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
        dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
    if (resField == NULL) {
        LOGD("DexOpt: couldn't find field %s.%s\n",
            resClass->descriptor,
            dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
        if (pFailure != NULL)
            *pFailure = VERIFY_ERROR_NO_FIELD;
        return NULL;
    }
    if (dvmIsStaticField(&resField->field)) {
        LOGD("DexOpt: wanted instance, got static for field %s.%s\n",
            resClass->descriptor,
            dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
        if (pFailure != NULL)
            *pFailure = VERIFY_ERROR_CLASS_CHANGE;
        return NULL;
    }

    /*
     * Add it to the resolved table so we're faster on the next lookup.
     */
    dvmDexSetResolvedField(pDvmDex, ifieldIdx, (Field*) resField);
}

/* access allowed? */
tweakLoader(referrer, resField->field.clazz);
bool allowed = dvmCheckFieldAccess(referrer, (Field*)resField);
```

The '104 Reissue Patent	Infringed By

```
    untweakLoader(referrer, resField->field.clazz);
    if (!allowed) {
        LOGI("DexOpt: access denied from %s to field %s.%s\n",
            referrer->descriptor, resField->field.clazz->descriptor,
            resField->field.name);
        if (pFailure != NULL)
            *pFailure = VERIFY_ERROR_ACCESS_FIELD;
        return NULL;
    }

    return resField;
}

/*
 * Alternate version of dvmResolveStaticField().
 *
 * Does not force initialization of the resolved field's class.
 *
 * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
 */
StaticField* dvmOptResolveStaticField(ClassObject* referrer, u4 sfieldIdx,
    VerifyError* pFailure)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    StaticField* resField;

    resField = (StaticField*)dvmDexGetResolvedField(pDvmDex, sfieldIdx);
    if (resField == NULL) {
        const DexFieldId* pFieldId;
        ClassObject* resClass;

        pFieldId = dexGetFieldId(pDvmDex->pDexFile, sfieldIdx);
```

The '104 Reissue Patent	Infringed By

```
/*
 * Find the field's class.
 */
resClass = dvmOptResolveClass(referrer, pFieldId->classIdx, pFailure);
if (resClass == NULL) {
    //dvmClearOptException(dvmThreadSelf());
    assert(!dvmCheckException(dvmThreadSelf()));
    if (pFailure != NULL) { assert(!VERIFY_OK(*pFailure)); }
    return NULL;
}

resField = (StaticField*)dvmFindFieldHier(resClass,
            dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
            dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
if (resField == NULL) {
    LOGD("DexOpt: couldn't find static field\n");
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_NO_FIELD;
    return NULL;
}
if (!dvmIsStaticField(&resField->field)) {
    LOGD("DexOpt: wanted static, got instance for field %s.%s\n",
        resClass->descriptor,
        dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_CLASS_CHANGE;
    return NULL;
}

/*
 * Add it to the resolved table so we're faster on the next lookup.
```

The '104 Reissue Patent	Infringed By

	*
	* We can only do this if we're in "dexopt", because the presence
	* of a valid value in the resolution table implies that the class
	* containing the static field has been initialized.
	*/
	if (gDvm.optimizing)
	dvmDexSetResolvedField(pDvmDex, sfieldIdx, (Field*) resField);
	}
	
	/* access allowed? */
	tweakLoader(referrer, resField->field.clazz);
	bool allowed = dvmCheckFieldAccess(referrer, (Field*)resField);
	untweakLoader(referrer, resField->field.clazz);
	if (!allowed) {
	LOGI("DexOpt: access denied from %s to field %s.%s\n",
	referrer->descriptor, resField->field.clazz->descriptor,
	resField->field.name);
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_ACCESS_FIELD;
	return NULL;
	}
	
	return resField;
	}
	
	/*
	* Rewrite an iget/iput instruction. These all have the form:
	* op vA, vB, field@CCCC
	*
	* Where vA holds the value, vB holds the object reference, and CCCC is
	* the field reference constant pool offset. We want to replace CCCC
	* with the byte offset from the start of the object.

The '104 Reissue Patent	Infringed By

```
 *
 * "clazz" is the referring class.  We need this because we verify
 * access rights here.
 */
static void rewriteInstField(Method* method, u2* insns, OpCode newOpc)
{
    ClassObject* clazz = method->clazz;
    u2 fieldIdx = insns[1];
    InstField* field;
    int byteOffset;

    field = dvmOptResolveInstField(clazz, fieldIdx, NULL);
    if (field == NULL) {
        LOGI("DexOpt: unable to optimize field ref 0x%04x at 0x%02x in %s.%s\n",
            fieldIdx, (int) (insns - method->insns), clazz->descriptor,
            method->name);
        return;
    }

    if (field->byteOffset >= 65536) {
        LOGI("DexOpt: field offset exceeds 64K (%d)\n", field->byteOffset);
        return;
    }

    insns[0] = (insns[0] & 0xff00) | (u2) newOpc;
    insns[1] = (u2) field->byteOffset;
    LOGVV("DexOpt: rewrote access to %s.%s --> %d\n",
        field->field.clazz->descriptor, field->field.name,
        field->byteOffset);
}

/*
```

The '104 Reissue Patent	Infringed By

```
 * Alternate version of dvmResolveMethod().
 *
 * Doesn't throw exceptions, and checks access on every lookup.
 *
 * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
 */
Method* dvmOptResolveMethod(ClassObject* referrer, u4 methodIdx,
    MethodType methodType, VerifyError* pFailure)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    Method* resMethod;

    assert(methodType == METHOD_DIRECT ||
           methodType == METHOD_VIRTUAL ||
           methodType == METHOD_STATIC);

    LOGVV("--- resolving method %u (referrer=%s)\n", methodIdx,
        referrer->descriptor);

    resMethod = dvmDexGetResolvedMethod(pDvmDex, methodIdx);
    if (resMethod == NULL) {
        const DexMethodId* pMethodId;
        ClassObject* resClass;

        pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

        resClass = dvmOptResolveClass(referrer, pMethodId->classIdx, pFailure);
        if (resClass == NULL) {
            /*
             * Can't find the class that the method is a part of, or don't
             * have permission to access the class.
             */
```

The '104 Reissue Patent	Infringed By

	LOGV("DexOpt: can't find called method's class (?.%s)\n",
	dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx));
	if (pFailure != NULL) { assert(!VERIFY_OK(*pFailure)); }
	return NULL;
	}
	if (dvmIsInterfaceClass(resClass)) {
	/* method is part of an interface; this is wrong method for that */
	LOGW("DexOpt: method is in an interface\n");
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_GENERIC;
	return NULL;
	}
	
	/*
	* We need to chase up the class hierarchy to find methods defined
	* in super-classes. (We only want to check the current class
	* if we're looking for a constructor.)
	*/
	DexProto proto;
	dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);
	
	if (methodType == METHOD_DIRECT) {
	resMethod = dvmFindDirectMethod(resClass,
	dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx), &proto);
	} else {
	/* METHOD_STATIC or METHOD_VIRTUAL */
	resMethod = dvmFindMethodHier(resClass,
	dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx), &proto);
	}
	
	if (resMethod == NULL) {
	LOGV("DexOpt: couldn't find method '%s'\n",

The '104 Reissue Patent	Infringed By

	dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx));
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_NO_METHOD;
	return NULL;
	}
	if (methodType == METHOD_STATIC) {
	if (!dvmIsStaticMethod(resMethod)) {
	LOGD("DexOpt: wanted static, got instance for method %s.%s\n",
	resClass->descriptor, resMethod->name);
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_CLASS_CHANGE;
	return NULL;
	}
	} else if (methodType == METHOD_VIRTUAL) {
	if (dvmIsStaticMethod(resMethod)) {
	LOGD("DexOpt: wanted instance, got static for method %s.%s\n",
	resClass->descriptor, resMethod->name);
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_CLASS_CHANGE;
	return NULL;
	}
	}
	
	/* see if this is a pure-abstract method */
	if (dvmIsAbstractMethod(resMethod) && !dvmIsAbstractClass(resClass)) {
	LOGW("DexOpt: pure-abstract method '%s' in %s\n",
	dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx),
	resClass->descriptor);
	if (pFailure != NULL)
	*pFailure = VERIFY_ERROR_GENERIC;
	return NULL;
	}

The '104 Reissue Patent	Infringed By

```
    /*
     * Add it to the resolved table so we're faster on the next lookup.
     *
     * We can only do this for static methods if we're not in "dexopt",
     * because the presence of a valid value in the resolution table
     * implies that the class containing the static field has been
     * initialized.
     */
    if (methodType != METHOD_STATIC || gDvm.optimizing)
        dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);
}

LOGVV("--- found method %d (%s.%s)\n",
    methodIdx, resMethod->clazz->descriptor, resMethod->name);

/* access allowed? */
tweakLoader(referrer, resMethod->clazz);
bool allowed = dvmCheckMethodAccess(referrer, resMethod);
untweakLoader(referrer, resMethod->clazz);
if (!allowed) {
    IF_LOGI() {
        char* desc = dexProtoCopyMethodDescriptor(&resMethod->prototype);
        LOGI("DexOpt: illegal method access (call %s.%s %s from %s)\n",
            resMethod->clazz->descriptor, resMethod->name, desc,
            referrer->descriptor);
        free(desc);
    }
    if (pFailure != NULL)
        *pFailure = VERIFY_ERROR_ACCESS_METHOD;
    return NULL;
}
```

The '104 Reissue Patent	Infringed By

```
    return resMethod;
}

/*
 * Rewrite invoke-virtual, invoke-virtual/range, invoke-super, and
 * invoke-super/range.  These all have the form:
 *   op vAA, meth@BBBB, reg stuff @CCCC
 *
 * We want to replace the method constant pool index BBBB with the
 * vtable index.
 */
static bool rewriteVirtualInvoke(Method* method, u2* insns, OpCode newOpc)
{
    ClassObject* clazz = method->clazz;
    Method* baseMethod;
    u2 methodIdx = insns[1];

    baseMethod = dvmOptResolveMethod(clazz, methodIdx, METHOD_VIRTUAL, NULL);
    if (baseMethod == NULL) {
        LOGD("DexOpt: unable to optimize virt call 0x%04x at 0x%02x in %s.%s\n",
            methodIdx,
            (int) (insns - method->insns), clazz->descriptor,
            method->name);
        return false;
    }

    assert((insns[0] & 0xff) == OP_INVOKE_VIRTUAL ||
           (insns[0] & 0xff) == OP_INVOKE_VIRTUAL_RANGE ||
           (insns[0] & 0xff) == OP_INVOKE_SUPER ||
           (insns[0] & 0xff) == OP_INVOKE_SUPER_RANGE);
```

The '104 Reissue Patent	Infringed By

```
    /*
     * Note: Method->methodIndex is a u2 and is range checked during the
     * initial load.
     */
    insns[0] = (insns[0] & 0xff00) | (u2) newOpc;
    insns[1] = baseMethod->methodIndex;

    //LOGI("DexOpt: rewrote call to %s.%s --> %s.%s\n",
    //    method->clazz->descriptor, method->name,
    //    baseMethod->clazz->descriptor, baseMethod->name);

    return true;
}


…
/*
 * Resolve an interface method reference.
 *
 * No method access check here -- interface methods are always public.
 *
 * Returns NULL if the method was not found.  Does not throw an exception.
 */
Method* dvmOptResolveInterfaceMethod(ClassObject* referrer, u4 methodIdx)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    Method* resMethod;
    int i;

    LOGVV("--- resolving interface method %d (referrer=%s)\n",
        methodIdx, referrer->descriptor);

    resMethod = dvmDexGetResolvedMethod(pDvmDex, methodIdx);
```

The '104 Reissue Patent	Infringed By

```
if (resMethod == NULL) {
    const DexMethodId* pMethodId;
    ClassObject* resClass;

    pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

    resClass = dvmOptResolveClass(referrer, pMethodId->classIdx, NULL);
    if (resClass == NULL) {
        /* can't find the class that the method is a part of */
        dvmClearOptException(dvmThreadSelf());
        return NULL;
    }
    if (!dvmIsInterfaceClass(resClass)) {
        /* whoops */
        LOGI("Interface method not part of interface class\n");
        return NULL;
    }

    const char* methodName =
        dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx);
    DexProto proto;
    dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);

    LOGVV("+++ looking for '%s' '%s' in resClass='%s'\n",
        methodName, methodSig, resClass->descriptor);
    resMethod = dvmFindVirtualMethod(resClass, methodName, &proto);
    if (resMethod == NULL) {
        /* scan superinterfaces and superclass interfaces */
        LOGVV("+++ did not resolve immediately\n");
        for (i = 0; i < resClass->iftableCount; i++) {
            resMethod = dvmFindVirtualMethod(resClass->iftable[i].clazz,
                    methodName, &proto);
```

The '104 Reissue Patent	Infringed By

	if (resMethod != NULL)
	break;
	}
	
	if (resMethod == NULL) {
	LOGVV("+++ unable to resolve method %s\n", methodName);
	return NULL;
	}
	} else {
	LOGVV("+++ resolved immediately: %s (%s %d)\n", resMethod->name,
	resMethod->clazz->descriptor, (u4) resMethod->methodIndex);
	}
	
	/* we're expecting this to be abstract */
	if (!dvmIsAbstractMethod(resMethod)) {
	char* desc = dexProtoCopyMethodDescriptor(&resMethod->prototype);
	LOGW("Found non-abstract interface method %s.%s %s\n",
	resMethod->clazz->descriptor, resMethod->name, desc);
	free(desc);
	return NULL;
	}
	
	/*
	* Add it to the resolved table so we're faster on the next lookup.
	*/
	dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);
	}
	
	LOGVV("--- found interface method %d (%s.%s)\n",
	methodIdx, resMethod->clazz->descriptor, resMethod->name);
	
	/* interface methods are always public; no need to check access */

The ’104 Reissue Patent	Infringed By

```
    return resMethod;
}

…
```

See also, e.g., dalvik\vm\analysis\DexOptimize.c (similarly in dalvik\vm\analysis\Optimize.c)::

```
/*
 * Optimize instructions in a method.
 *
 * Returns "true" if all went well, "false" if we bailed out early when
 * something failed.
 */
static bool optimizeMethod(Method* method, const InlineSub* inlineSubs)
{
    u4 insnsSize;
    u2* insns;
    u2 inst;

    if (dvmIsNativeMethod(method) || dvmIsAbstractMethod(method))
        return true;

    insns = (u2*) method->insns;
    assert(insns != NULL);
    insnsSize = dvmGetMethodInsnsSize(method);

    while (insnsSize > 0) {
        int width;

        inst = *insns & 0xff;

        switch (inst) {
```

The '104 Reissue Patent	Infringed By

	case OP_IGET:
	case OP_IGET_BOOLEAN:
	case OP_IGET_BYTE:
	case OP_IGET_CHAR:
	case OP_IGET_SHORT:
	rewriteInstField(method, insns, OP_IGET_s);
	break;
	case OP_IGET_WIDE:
	rewriteInstField(method, insns, OP_IGET_WIDE_QUICK);
	break;
	case OP_IGET_OBJECT:
	rewriteInstField(method, insns, OP_IGET_OBJECT_QUICK);
	break;
	case OP_IPUT:
	case OP_IPUT_BOOLEAN:
	case OP_IPUT_BYTE:
	case OP_IPUT_CHAR:
	case OP_IPUT_SHORT:
	rewriteInstField(method, insns, OP_IPUT_QUICK);
	break;
	case OP_IPUT_WIDE:
	rewriteInstField(method, insns, OP_IPUT_WIDE_QUICK);
	break;
	case OP_IPUT_OBJECT:
	rewriteInstField(method, insns, OP_IPUT_OBJECT_QUICK);
	break;
	
	case OP_INVOKE_VIRTUAL:
	if (!rewriteExecuteInline(method, insns, METHOD_VIRTUAL,inlineSubs))
	{
	if (!rewriteVirtualInvoke(method, insns, OP_INVOKE_VIRTUAL_QUICK))
	return false;

The '104 Reissue Patent	Infringed By

	}
	break;
	case OP_INVOKE_VIRTUAL_RANGE:
	if (!rewriteExecuteInlineRange(method, insns, METHOD_VIRTUAL,
	inlineSubs))
	{
	if (!rewriteVirtualInvoke(method, insns,
	OP_INVOKE_VIRTUAL_QUICK_RANGE))
	{
	return false;
	}
	}
	break;
	case OP_INVOKE_SUPER:
	if (!rewriteVirtualInvoke(method, insns, OP_INVOKE_SUPER_QUICK))
	return false;
	break;
	case OP_INVOKE_SUPER_RANGE:
	if (!rewriteVirtualInvoke(method, insns, OP_INVOKE_SUPER_QUICK_RANGE))
	return false;
	break;
	
	case OP_INVOKE_DIRECT:
	if (!rewriteExecuteInline(method, insns, METHOD_DIRECT, inlineSubs))
	{
	if (!rewriteEmptyDirectInvoke(method, insns))
	return false;
	}
	break;
	case OP_INVOKE_DIRECT_RANGE:
	rewriteExecuteInlineRange(method, insns, METHOD_DIRECT, inlineSubs);
	break;

The '104 Reissue Patent	Infringed By

```
        case OP_INVOKE_STATIC:
            rewriteExecuteInline(method, insns, METHOD_STATIC, inlineSubs);
            break;
        case OP_INVOKE_STATIC_RANGE:
            rewriteExecuteInlineRange(method, insns, METHOD_STATIC, inlineSubs);
            break;

        default:
            // ignore this instruction
            ;
        }

        if (*insns == kPackedSwitchSignature) {
            width = 4 + insns[1] * 2;
        } else if (*insns == kSparseSwitchSignature) {
            width = 2 + insns[1] * 4;
        } else if (*insns == kArrayDataSignature) {
            u2 elemWidth = insns[1];
            u4 len = insns[2] | (((u4)insns[3]) << 16);
            width = 4 + (elemWidth * len + 1) / 2;
        } else {
            width = dexGetInstrWidth(gDvm.instrWidth, inst);
        }
        assert(width > 0);

        insns += width;
        insnsSize -= width;
    }

    assert(insnsSize == 0);
    return true;
```

The '104 Reissue Patent	Infringed By
	}

The '104 Reissue Patent	Infringed By
12. A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting intermediate form object code comprised of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:	The Accused Instrumentalities include devices that store, distribute, or run Android or the Android SDK, including websites, servers, and mobile devices. They encompass a computer readable medium containing instructions for controlling a data processing system to perform a method for interpreting intermediate form object code comprised of instructions, certain of said instructions containing one or more symbolic references, to perform the steps described in the claim. *See* Claim 11, *supra*.
interpreting said instructions in accordance with a program execution control;	*See* Claim 11, *supra*. The Android platform has a Dalvik virtual machine that interprets intermediate form object code. Dexopt is part of the bytecode interpretation process because it's a pre-pass made over the bytecodes to facilitate optimized bytecode execution. *See, e.g.*, dalvik\docs\dexopt.html; *see also*, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html: …. **dexopt** We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in. The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files